A model-based approach to service creation

Dick A.C. Quartel, Marten J. van Sinderen, Luís Ferreira Pires
Centre for Telematics and Information Technology, University of Twente,
PO Box 217, 7500 AE Enschede, the Netherlands
e-mail: {quartel, sinderen, pires}@cs.utwente.nl

Abstract

This paper presents a model-based approach to support service creation. In this approach, services are assumed to be created from (available) software components. The creation process may involve multiple design steps in which the requested service is repeatedly decomposed into more detailed functional parts, until these parts can be mapped onto software components. A modelling language is used to express and enable analysis of the resulting designs, in particular the behaviour aspects. Methods are needed to verify the correctness of each design step. A technique called behaviour refinement is introduced to assess the conformance relation between an abstract behaviour and a more concrete (detailed) behaviour. This technique is based on the application of abstraction rules to determine the abstraction of the concrete behaviour, such that the obtained abstraction can be compared to the original abstract behaviour. The application of this refinement technique throughout the creation process enforces the correctness of the created service.

1 Introduction

Currently, much research takes place on (software) architectures, methods and techniques that enable the fast introduction of new telematics applications. For example, middleware platforms and component models have been developed to provide software abstractions to application developers in order to facilitate the rapid introduction of new applications. Middleware platforms enable the interaction between distributed software components, abstracting from the complexity and heterogeneity of the underlying networks and operating systems. Component models facilitate the development and reuse of software components providing generic pieces of functionality, enabling applications to be built from deployable building blocks that can be considered at high abstraction levels ([8]).

Assuming the availability of middleware platforms, component models and libraries, the development of new applications is reduced to composing software components in such a way that the requested functionality is provided. However, the following problems still have to be solved:

1. how to find a proper (de-)composition of components, making optimal re-use of available software components;
2. how to determine that the chosen composition of components is correct, i.e., provides the requested functionality.

Re-use of components may speed up the development process of new applications. Re-use of components is, however, not straightforward. A decomposition of the requested application into available components may be difficult to find (if possible at all), or alternative decompositions may exist. Therefore, guidelines are needed for the decomposition of applications into components as well as techniques to support the selection and search of components.

Correctness is important for all applications (but for some more than for others). Techniques are needed to assess the correctness of components and their cooperation. Examples of such techniques are simulation and model checking, which are applied at design time, and testing of pre- and post-conditions on operations, which is applied at run-time.

The process of creating a new service that is provided by an application composed from software components is called *service creation* furtheron. As a result of this process, software components may be specified that are not available yet. The implementation of such software components falls outside the scope of the service creation process.

The objective of this paper is to present a model-based approach to service creation. This approach involves techniques to model and analyse software components and their compositions. In particular, we focus on a technique to assess the conformance between the requested service and its implementation consisting of a composition of software components. This technique provides a solution to the second problem mentioned above.

The remainder of this paper is structured as follows: section 2 describes the service creation process and motivates our modelling approach, section 3 discusses techniques to model software components, section 4 describes a technique to perform conformance assessment, section 5 applies the techniques to an example, section 6 discusses abstraction from communication aspects between components, and section 7 presents some conclusions.

0-7695-0468-X/99 $10.00 © 1999 IEEE

2 Service creation process

A *service* is defined as the external behaviour of a system. Examples of systems in the context of this paper are a distributed computing system, a telematics application and a software component. The external behaviour of such systems consists of the functions provided by these systems to their environment (users), and the relationships between these functions. Furthermore, non-functional aspects, e.g., performance aspects, may be included, such as timing and reliability. A service abstracts from how the external behaviour is provided, i.e., abstracts from any internal system aspects.

The following major milestones are distinguished in the service creation process:

1. *Specification* of the requested service;
2. *Design* of the requested service using (available) software components;
3. *Verification* of the design against the specification;
4. *Assembly* and packaging of the software components into a single, compound software component;
5. *Testing* the new software component;
6. *Deployment*.

Role of modelling. Our model-based approach focuses on the first three steps. Figure 1 depicts these steps, and their relationships. The service creation process starts with a specification of the service, derived from the user requirements. Techniques supporting requirements analysis, such as use cases, can be used in this step, but are not considered in this paper.

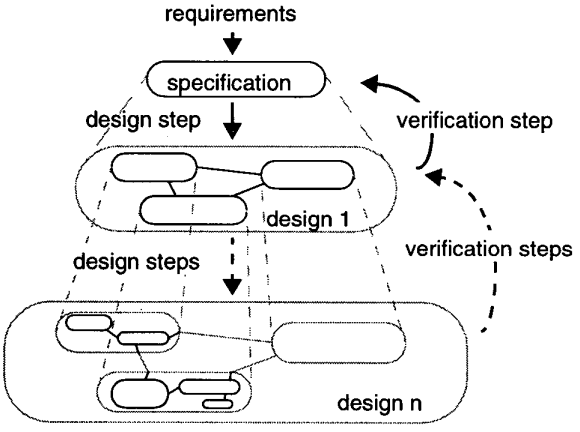

Figure 1: Specification, design and verification

We assume that the requested service has to be provided by a (compound) software component. Initially, this component is considered as a whole. Furthermore, we assume that no component is available that provides a service similar to the requested service. Otherwise, the service creation process would be reduced to some minor customization steps.

In the design step, the requested component is decomposed, possibly repeatedly, into multiple related functional parts, until a mapping of these functional parts onto existing or implementable software components is possible. In case component reuse is possible for a large part of the design, the time duration of this step is mainly determined by available knowledge to find these components and the skills to incorporate them effectively in the design. Modelling helps here in two ways: (i) the identification and search for components can be based on models representing the relevant characteristics of components, and (ii) the modelling of compositions of components allows one to assess whether a composition provides the requested service. Section 3 presents techniques to model components from a service perspective and from the perspective of a composition of functional parts.

Role of conformance assessment. The use of a model-based approach allows various types of verification and testing. This paper focuses on verifying the conformance relationship between different models of a system at related abstraction levels. Section 4 presents a technique to assess whether an internal model of some system correctly implements an external model of the same system. This technique can be applied after each design step. In case conformance assessment fails for some design step, this step has to be reconsidered, making service creation an iterative process.

Gap between design and deployment. Given a design of the requested service in terms of a set of software components, and their relationships, its packaging into a new compound component can be largely automated. Only in case some of the required components are not available, software development is needed. However, the service specification of such a component can be used to derive, e.g., IDL specifications or message sequences to help the component developer. Furthermore, some code or scripts to link components may have to be written manually, depending on the complexity of the relationships between components.

A model-based approach also supports testing. For example, tests can be derived automatically from the model of some component, and be compared with executions of this component in operation.

Modelling language and environment. A single modelling language, called Amber, is used. For an explanation of the architectural concepts underlying this language, we refer to [6] and [5]. The language is embedded in an integrated tool environment, called Testbed Studio[1], supporting the representation of models, some analysis techniques and documentation [3].

3 Modelling

An Amber model of some system consists of two sub-models: an entity model and a behaviour model.

3.1 Entity model

The entity model represents which system parts are considered, and how they are interconnected. Two concepts are used here: entity and interaction point. An *entity* represents a system (part) that performs some function or behaviour. For example, a component can be modelled as an entity. An *interaction point* represents some mechanism through which an entity can interact with other entities. For example, a (remote) procedure call mechanism and an Object Request Broker (ORB) can be modelled as an interaction point.

From the external perspective, a system is modelled by a single entity, having one or more interaction points. Figure 2(i) depicts the entity model of a component, called *C*, having two interaction points, called *IP1* and *IP2*, from an external perspective. An entity is graphically expressed by a rectangle with cut-off corners. An interaction point is graphically expressed by an ellipsis that overlaps with the entities that share the interaction point. In this case, the environment of the component is not modelled.

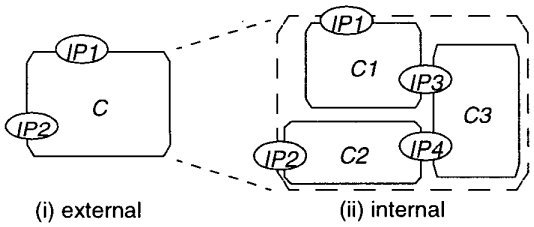

Figure 2: Entity model of a component

From the internal perspective, a system is modelled as a composition of functional parts. For example, these parts may represent objects or sub-components. Figure 2(ii) depicts an entity model of component *C* from an internal perspective. Component *C* is composed of three functional parts *C1*, *C2* and *C3*, such that parts *C1* and *C3* and parts *C2* and *C3* can interact directly via interaction points *IP3* and *IP4*, respectively, and parts *C1* and *C2* implement the interfaces at interaction points *IP1* and *IP2*, respectively.

3.2 Behaviour model

The behaviour model represents the behaviour, or functionality, of each entity in the corresponding entity model. Three concepts are used: action, interaction and causality relation. An *action* represents some unit of activity performed by a *single* entity. For example, the presentation of an HTML page by a Web browser can be modelled as an action. An *interaction* represents a common activity performed by *two (or more)* entities. For example, a method invocation can be modelled as an interaction between the invoking object and invoked object. *Information*, *time* and *location attributes* can be added to an (inter)action, in order to model the result established in some activity, the time moment at which this result is available, and the location where the result is available, respectively.

An (inter)action occurrence represents the successful completion of an activity. In case an (inter)action occurs, the same result is established and made available at the same time moment and at the same location for all entities involved in the activity, otherwise no result is established and no entity can refer to any intermediate results of the activity. An (inter)action abstracts from how an activity result is established. In order to model the latter, the activity should be decomposed into sub-activities, such that they can be modelled by distinct (inter)actions.

Figure 3(i) and (ii) depict an action *a* and an interaction $\underline{a}$, respectively. An action is graphically expressed as a circle (or ellipsis). An interaction is graphically expressed as a segmented circle (or ellipsis), which reflects that multiple entities contribute to the interaction. Interaction names are underlined in order to distinguish them from action names. The information (ι), time (τ) and location (λ) attributes are represented within a textbox attached to the (inter)action[1]. Constraints can be defined on the possible outcomes of the values of ι, τ and λ (after the symbol 'I'). In case of an interaction, each involved entity can define its constraints, such that the values of ι, τ and λ must satisfy all constraints, otherwise the interaction can not happen. In case multiple values are possible for some attribute, a non-deterministic choice between these values is assumed. In this example, interaction $\underline{a}$ is a decomposition (refinement) of action *a*, allowing the same attribute values to be established.

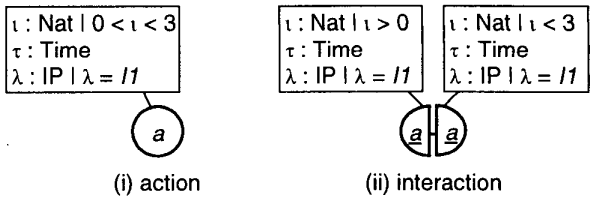

Figure 3: Action and interaction

A *causality relation* is associated with each (inter)action, modelling the conditions for this (inter)action to happen.

[1]Currently, Testbed Studio supports a simplified version of Amber, using a specialized syntax, tailored to model business processes. Amber itself is more generic, supporting the modelling of distributed systems, of which business processes are considered a specific case.

[1]An intuitive notation is used here to represent attribute values, types and constraints.

Three basic conditions for the occurrence of some action *a* are identified:

- $b \rightarrow a$; action *b* must happen before action *a*;
- $\neg b \rightarrow a$; action *b* must not happen before nor simultaneously with action *a*;
- $^{=}b \rightarrow a$; action *b* must happen simultaneously with *a*.

The and- ($\wedge$) and or-operator ($\vee$) can be used to model more complex causality conditions. For example, $b \vee \neg c \rightarrow a$ represents that action *a* can happen after action *b* has happened or as long as action *c* has not happened yet. Furthermore, a probability attribute can be added to each sufficient condition to model the probability the (inter)action happens when this condition is satisfied.

The causality relation concept allows the modelling of many different relationships between actions. It is often more convenient to express these relationships directly, instead as a composition of causality relations of the individual actions. Figure 4 depicts the graphical expressions of some common relationships between two or three actions. The $\wedge$- and $\vee$-operator are graphically expressed by the symbols ■ and □, respectively.

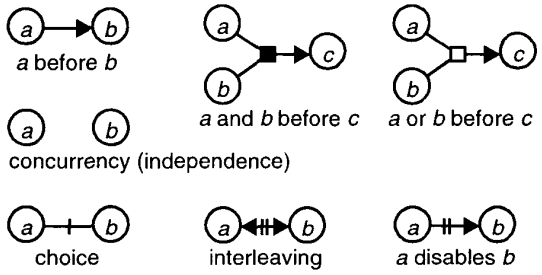

Figure 4: Common action relations

A full explanation of the Amber language is beyond the scope of this paper. However, sufficient explanation is added to the examples below to enable one to follow the line of reasoning without elaborate knowledge of Amber.

External behaviour. The external behaviour of a system defines the interactions with the system's environment, and their relationships. In case of a component, the interactions are the operations that can be invoked on this component and the operations that can be invoked by this component on other components.

Operations. An operation is a function provided by some component that can be invoked by other components. Two types of operations are distinguished: an *interrogation*, which returns a result to the invoking component, and an *announcement*, which does not return a result to the invoking component.

Figure 5(i) depicts the modelling of an announcement as a single interaction, called *inv*, between two components. The rounded rectangles, called *C1* and *C2*, represent the behaviours of the invoking and invoked component, respectively. An arrow pointing to an (inter)action that is not connected to any other (inter)action, represents that the (inter)action is allowed to occur from the beginning of the behaviour.

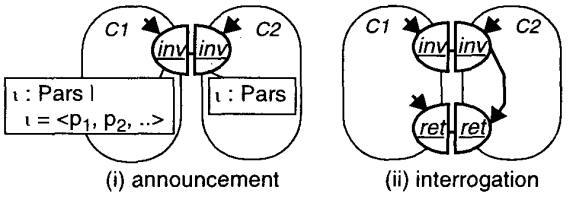

Figure 5: Operations

Figure 5(ii) depicts the modelling of an interrogation as the sequential composition of two interactions *inv* and *ret*, modelling the invocation and the return of the result of the operation, respectively. In this case, the causality between *inv* and *ret* is made the responsibility of *C2*. Similar to Figure 5(i), the information attribute can be used to model the information values passed in the *inv* interaction and the result returned in the *ret* interaction. In addition, value and type constraints can be defined on the operation parameters.

Interfaces. The operations invoked by/on a component are generally structured into interfaces. An interface can be modelled as a sub-behaviour of a component's behaviour, defining a subset of the operations and their relationships. Furthermore, one can define a distinct interaction point for each interface in the entity model.

Internal behaviour. The internal behaviour of a component can be modelled in the following ways:

1. by adding internal (inter)actions, and their relationships, to the external behaviour description;
2. as a composition of the external behaviours of the functional parts in which the component is decomposed;
3. a combination of both.

Figure 6(i) depicts an example of the first option. The external behaviour of component *C2* in Figure 5 is refined with two actions *c* and *d* modeling two internal sub-activities in which (part of) the result is established that is returned in interaction *ret*.

Figure 6(ii) depicts an example of the second option. Behaviour *C2* in Figure 5 is decomposed into a behaviour *S* responsible for accepting an invocation and returning a result, and a behaviour *DB* performing, e.g., some database function used by *S* to produce the operation result. Behaviours S and DB can be assigned to different components.

Component bindings. So far we have assumed that components can interact directly, such that the mechanisms supporting the binding between components, such as an ORB, do not have to be modelled explicitly. Section 6 shows that

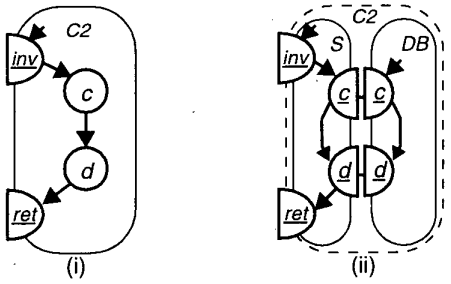

Figure 6: Adding internal behaviour

this abstraction is only correct if the ORB is reliable; otherwise ORB unreliability should be modelled explicitly.

Performance aspects. Time and probability attributes can be used to model performance aspects. Figure 7 depicts an example in which a maximal response time and a maximal failure probability is imposed on an operation. For convenience, the operation invocation and operation return are modelled as actions, abstracting from the entities involved. The time constraint $\tau_{ret} = \tau_{inv} + 5$ defines that interaction *ret* must occur within 5 time units after the time moment (τ_{inv}) at which interaction *inv* has occurred. The probability constraint $\pi_{ret/inv} \geq 0.9$ defines the conditional probability that interaction *ret* occurs after interaction *inv* has occurred; in this case, given 100 behaviour executions in which interaction *inv* occurs, interaction *ret* must occur on the average in at least 90 of these executions. The next section presents an alternative model of an interrogation that returns an exception in case of failure.

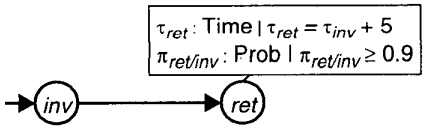

Figure 7: Modelling delay and reliablility

4 Conformance assessment

In the course of a design process, abstract designs have to be replaced by more concrete designs. An abstract design prescribes *what* should be implemented, whereas a concrete design prescribes *how* this abstract design should be implemented. The notions of abstract design and concrete design are relative, since a more concrete design can be considered as an abstract design in a next design step.

For example, in Figure 1 the external specification of some component can be considered an abstract design, which prescribes what service should be provided by this component. The internal specification of some component as a composition of functional parts can be considered a concrete design, which prescribes how the service is provided. The external and internal perspective can be applied recursively to the functional parts, illustrating the relative notion of abstract and concrete design.

This section presents a technique to enforce the correct replacement of an abstract behaviour by a concrete behaviour, called *behaviour refinement*. We focus on behaviour, since it comprises most of the complexity of a design.

Behaviour refinement. The objective of behaviour refinement is to replace an abstract behaviour by a more concrete behaviour that conforms to this abstract behaviour. Figure 8 depicts an example of behaviour refinement. Behaviours *B1* and *B2* model an interrogation, such that *B2* is a refinement of *B1*. Action *prc* models the processing of the invocation, action *tio* models a time-out that happens when the processing can not be finished in time. Action *exc* represents the generation of an exception after a time-out occurs. Both *B1* and *B2* are refinements of the original model of an interrogation presented in Figure 5(ii); this is explained later on. Attribute type definitions have been omitted for brevity.

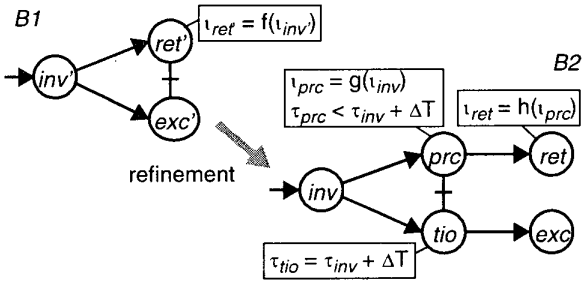

Figure 8: Behaviour refinement

Actions of abstract behaviours are called *abstract actions* and actions of concrete behaviours are called *concrete actions*. We assume that the occurrence of each abstract action corresponds to the occurrence of one or more concrete actions. This assumption makes it possible to compare the abstract behaviour with the concrete behaviour, by comparing the abstract actions with their corresponding concrete actions. This comparison is needed in order to assess whether the concrete behaviour conforms to the abstract behaviour.

Concrete actions that correspond to abstract actions are called *reference actions*, since they are considered as reference points in the concrete behaviour for assessing conformance. Concrete actions that are not reference actions are called *inserted actions*, since they are inserted during behaviour refinement. For example, in Figure 8 actions *inv'* and *ret'* are abstract actions and actions *inv*, *tio*, *prc*, *exc* and *ret* are concrete actions. Actions *inv* and *ret* are reference actions that correspond to abstract actions *inv'* and *ret'*. Actions *tio*, *prc* and *exc* are inserted actions. Abstract actions are denoted by the action identifiers of their corresponding reference actions appended with a prime.

A *conformance relation* defines which concrete behaviours are valid refinements, or implementations, of the abstract behaviour. This conformance relation should guarantee that what is prescribed in the abstract behaviour is preserved by the concrete behaviour. The following requirements for conformance are identified:

1. *preservation of action relations*: the structure of relations between abstract actions is preserved by the structure of relations between their corresponding concrete actions;
2. *preservation of attribute values*: attribute values of abstract actions are preserved by the attributes of their corresponding concrete actions;
3. *preservation of attribute value relations*: relations between attribute values of abstract actions are preserved by the relations between the attributes of the corresponding concrete actions.

For example, in Figure 8 the ordering relation between abstract actions *inv'* and *ret'* is preserved by the conjunction of the ordering relation between concrete actions *inv* and *prc* and the ordering relation between concrete actions *prc* and *ret*. The choice relation between abstract actions *ret'* and *exc'* is preserved by the choice relation between concrete actions *tio* and *prc*. The information reference relation $\iota_{ret'} = f(\iota_{inv'})$ between abstract actions *inv'* and *ret'* is preserved by the conjunction of information reference relation $\iota_{prc} = g(\iota_{inv})$ between concrete actions *prc* and *inv* and information reference relation $\iota_{ret} = g(\iota_{prc})$ between concrete actions *ret* and *prc*, assuming $f = g \cdot h$.

Basic types of refinement. Two basic types of behaviour refinement are distinguished:

- *causality refinement*, which consists of replacing causality relations between abstract actions by causality relations involving their corresponding concrete actions and some inserted actions;
- *action refinement*, which consists of replacing an abstract action by a concrete activity involving multiple concrete actions and their causality relations.

Instances of behaviour refinement may consist of one of these basic types of refinement or a combination of both. The essential difference between causality refinement and action refinement is the way attributes of abstract actions are distributed over the attributes of concrete actions. This difference is reflected in distinct specializations of the preservation of attribute values conformance requirement, one for each basic type of behaviour refinement.

Causality refinement allows one to model the relations between abstract actions in more detail through the introduction of inserted actions. Inserted actions model additional activities in the concrete behaviour that were not relevant during the definition of the abstract behaviour. An essential characteristic of causality refinement is that the attributes of an abstract action are preserved by the attributes of a single concrete action. Therefore, each abstract action corresponds to a *single* reference action.

Action refinement allows one to model an activity that is represented by a single abstract action in more detail. The activity is decomposed into multiple related sub-activities that are represented by concrete actions and their causality relations. An essential characteristic of action refinement is that at least one of the attributes of the abstract action is distributed over the attributes of multiple concrete actions in the activity.

Figure 9 depicts an example of action refinement and causality refinement. Abstract action *rsl"* (*result*) can establish two possible information values: $f(\iota_{inv''})$ and "exc", where "exc" represents the occurrence of an exception in terms of data. This action is refined into a choice between concrete (final) actions *ret'* and *exc'*, such that *ret'* establishes the information value $f(\iota_{inv''})$ and *exc'* represents the occurrence of an exception. The refinement of *rsl"* is an example of action refinement, since the possible results of *rsl"* are modelled as two different final actions.

The ordering relation between abstract actions *ret"* and *inv"* is defined such that *ret"* may not occur in some behaviour executions after *inv"* has occurred. This uncertainty is made explicit in the concrete behaviour by the occurrence of inserted action *exc'*, since according to our causality relations semantics, either *ret'* or *exc'* must occur after *inv'* has occurred. Furthermore, only the ordering relation is refined, since concrete action *ret'* corresponds to abstract action *ret"*, characterizing a causality refinement.

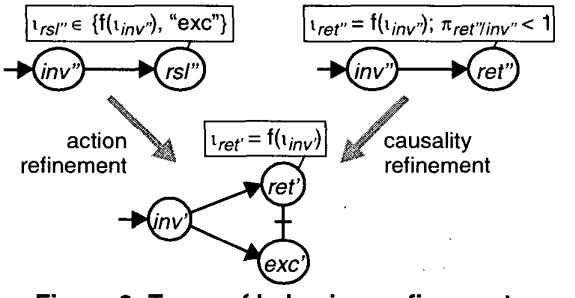

Figure 9: Types of behaviour refinement

Use of abstraction. An abstract behaviour can be replaced by many alternative concrete behaviours. Depending on the choice of a concrete behaviour, different concrete actions and their causality relations are added to the abstract behaviour. Since this choice is determined by specific design objectives, behaviour refinement can not be automated in its totality.

When abstracting from certain concrete actions and their causality relations, the abstraction of this concrete behaviour is completely determined by the remaining concrete actions and their causality relations. In these circumstances, the abstraction of a concrete behaviour is unique. Rules can

be provided to calculate this abstraction. These rules can, in principle, be automated.

The uniqueness of an abstraction allows one to assess the conformance between an abstract behaviour and a concrete behaviour, by comparing the abstraction of the concrete behaviour with the original abstract behaviour. Therefore, we distinguish the following subsequent design activities in an instance of behaviour refinement:

1. *delimitation of the abstract behaviour*: we only consider the refinement of behaviours that are influenced by a finite number of abstract actions. For example, in case of recursive behaviours one should identify the finite behaviour parts that are (infinitely) repeated;
2. *refinement of the abstract behaviour into a concrete behaviour*: in this activity we determine how the abstract behaviour is implemented by the concrete behaviour;
3. *determination of the abstraction of the concrete behaviour*: a method to perform this activity is outlined below;
4. *comparison of the abstraction of the concrete behaviour with the original abstract behaviour*: both behaviours should comply to a certain correctness relation, e.g., an equivalence relation. If this is not the case, the concrete behaviour is not considered a correct implementation of the abstract behaviour. In this case the designer must return to design activity 2.

Abstraction method. The following steps define a method to determine the abstraction of a concrete behaviour:

1. *identify reference actions and inserted actions in the concrete behaviour*: particularly, identified reference actions have to be considered as:
   - (single) reference actions, which are obtained when causality refinement has been applied; or
   - groups of reference actions, which are formed by grouping the final actions of each activity that is obtained when action refinement has been applied;
2. *abstract from inserted actions*: a complete method has been developed, consisting of concrete steps and rules that have to be followed to abstract from a single inserted action. This method only needs to consider the direct causality context of an inserted action *i*, consisting of *i* itself and the actions to which *i* is directly related. The abstraction of multiple inserted actions can be performed by consecutively abstracting from each single inserted action in any order. For example, independent of the order in which actions *prc* and *tio* in Figure 8 are abstracted from, the same abstract behaviour is obtained;
3. *replace each group of reference actions by an abstract action*: a complete method has been developed to abstract from an activity that replaces an abstract action and makes its attribute values available through the occurrence of one or more of its final actions. These final actions are the reference actions that correspond to the abstract action. The method distinguishes between the cases of a single final action, a conjunction of final actions, a disjunction of final actions (e.g., see Figure 9), or a combination of these. Rules are developed to obtain the causality relation and attribute definitions of the abstract action by integrating the causality conditions and attribute definitions of the final actions.

We refer to [5] and [7] for an explanation of the abstraction methods mentioned above.

5 Example: shared white board

Specification. Figure 10 depicts a simplified service specification of a shared white board application. A *d*raw operation (*d"*) can be requested by one of the participants, which is followed by either a *r*ejection (*r"*) indicated to this participant or an *u*pdate (*u"*) of the white board indicated to all participants. Action *d"* has two information (sub-)attributes $\iota 1_{d"}$: S and $\iota 2_{d"}$: O, where data types S and O represent the status of the shared white board and an operation to be performed on this status, respectively. The information attribute constraint $\iota_{r"} = \iota 1_{d"}$ of action *r"* represents that the status remains unchanged in case of a rejection. The information attribute constraint $\iota_{u"} = \iota 2_{d"}(\iota 1_{d"})$ of action *u"* represents that operation $\iota 2_{d"}$ is applied to the shared white board status.

A new draw action can only occur after action *r"* or action *u"* has occurred. Recursion is expressed by drawing a new instance of behaviour *Swb"*. An entry point (symbol $\triangleright$) is used to link the new behaviour instance to the remainder of the behaviour, such that the causality condition for a new instance of action *d"*, called $d"^{+}$, is: $u" \vee r" \rightarrow d"^{+}$. Furthermore, a parameter q : S is associated with the entry point to pass the information value established in *u"* or *r"* to the new instance of *d"*. Parameter q passes the value $\iota_{u"}$ in case *u"* occurs, and passes the value $\iota_{r"}$ in case *r"* occurs.

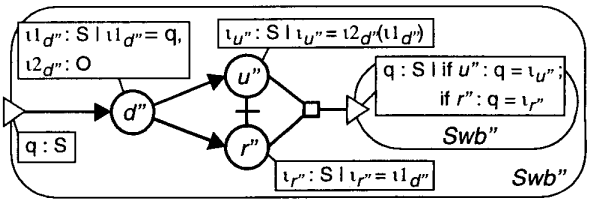

Figure 10: Requested service

Design. Figure 11 depicts a decomposition of the shared white board service into three parts:

- a client *C*, which requests server *S* to perform a draw operation, which is followed either by a rejection or by a *n*otification that the operation is processed (interaction *n*). After a notification, an *u*pdate of the shared white

board status is retrieved from database D. A new draw operation can be requested after a rejection or after an update has occurred;

- a server S, which handles draw operations. A draw request is followed either by a rejection to the originator of the request or by the processing of the draw operation (action p) and a notification to all participants that the new shared white board status can be retrieved from database D. Suppose that the designer inadvertently allows action p and interaction $\underline{n}$ to be performed in parallel. When action p is finished, the new shared white board status is stored into database D (interaction $\underline{s}$). A new draw request can be handled after interaction $\underline{r}$ or after both interactions $\underline{n}$ and $\underline{s}$ have occurred;
- a database D, which repeatedly either stores a new shared white board status or returns the currently stored status.

The complete status of the shared white board is maintained in the clients, and is passed via each draw and update operation. For the purpose of this example, the design only considers a single client.

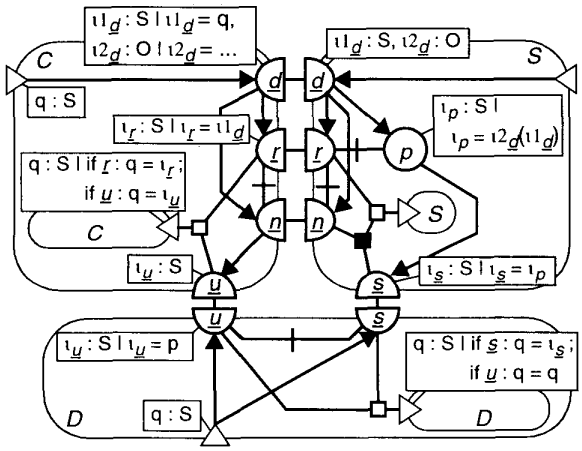

Figure 11: Design of shared white board

Verification. Figure 12 depicts the behaviour of the shared white board design of Figure 11, called *Swb*, while abstracting from the parts involved. The causality conditions of actions d, r, n, u and s are derived from interactions $\underline{d}$, $\underline{r}$, $\underline{n}$, $\underline{u}$ and $\underline{s}$, respectively, by calculating the conjunction of the causality conditions of the interaction parts in Figure 11. For example, the causality condition of a new instance of action d is equal to the conjunction of condition $(\underline{n} \wedge \underline{s}) \vee \underline{r}$ defined in behaviour S and condition $\underline{u} \vee \underline{r}$ in behaviour C, which renders condition $(\underline{n} \wedge \underline{s} \wedge \underline{u}) \vee (\underline{n} \wedge \underline{s} \wedge \underline{r}) \vee (\underline{u} \wedge \underline{r}) \vee \underline{r}$. This condition can be simplified to $(\underline{s} \wedge \underline{u}) \vee \underline{r}$, since the occurrence of $\underline{u}$ implies the occurrence of $\underline{n}$ and the occurrence of $\underline{r}$ conflicts with the occurrences of $\underline{n}$, $\underline{s}$ and $\underline{u}$. The interleaving relation between actions s and u is defined by behaviour D, which allows store and update operations to be performed in arbitrary order. Due to this interleaving relation, the information attribute of action u either refers via actions n and d to the old white board status (value $\iota 1_d$) in case u occurs before s, or refers to the processed white board status (value $\iota_s = \iota_p = \iota 2_d(\iota 1_d)$) in case s occurs before u.

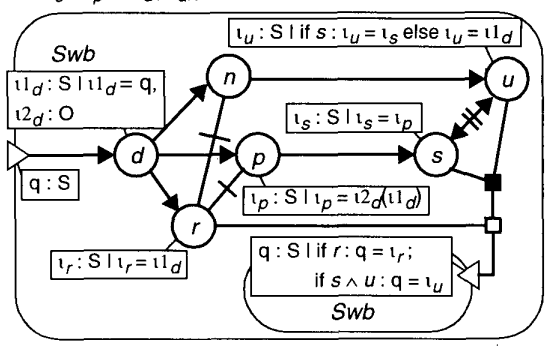

Figure 12: Concrete behaviour

Figure 13 depicts the abstraction of behaviour *Swb*, called *Swb'*. Concrete actions d, u and r are the reference actions corresponding to abstract actions d', u' and r', respectively. Actions p, n and s are inserted actions, which can be abstracted from in any order. Observe that the abstraction preserves the action relations and the information value relations between the reference actions.

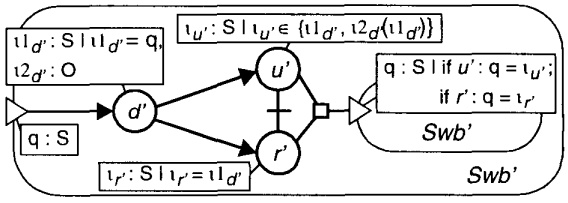

Figure 13: Abstract behaviour

Behaviour *Swb'* is not equivalent to the requested service *Swb"* in Figure 10. Action u' can establish information value $\iota 1_{d'}$, representing the loss of the preceding draw operation. This loss is caused by the parallel execution of action p and interaction $\underline{n}$ in server S (see Figure 11), allowing client C to update its shared white board status before the new status is stored. Consequently, concrete behaviour *Swb* does not conform to abstract behaviour *Swb"*, assuming behaviour equivalence as correctness relation.

This example shows that it is important to model the relationships between operations in order to analyse the behaviour of (compositions of) software components. Current Interface Description Languages (IDLs) do not support this requirement.

6 Abstracting from communication

Until now we have modelled operation invocations and returns as atomic interactions. This section discusses this modelling choice, by determining the conditions which guarantee that this abstract representation of invocations

and responses correctly models the communication support offered by an ORB.

Figure 14 depicts the abstraction/refinement relationship between the abstract model of an operation invocation used so far, and a more concrete model in which the behaviour responsible for supporting the communication between components, in this case an ORB, is explicitly represented. In Figure 14, i_{req} models the request to the ORB to deliver an operation invocation, and i_{ind} models the actual delivery of the invocation. Abstract action *inv'* corresponds to concrete action i_{ind}, such that *inv'* occurs if and only if i_{ind} occurs.

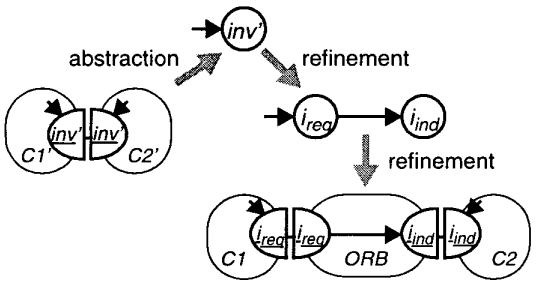

Figure 14: Abstraction from communication

Since the interaction $\underline{inv'}$ between components *C1'* and *C2'* in Figure 14 either occurs for both components or does not occur at all, it is fair to say that after $\underline{i_{req}}$ happens $\underline{i_{ind}}$ *has to happen* in the more concrete model. This implies that this modelling choice is only accurate if the ORB supports reliable communication. Furthermore, the behaviour of *C2* should be defined such that $\underline{i_{ind}}$ is not disabled after $\underline{i_{req}}$ happens (disabling would cause the interaction to happen only at the *C1* side, and consequently only at the *C1'* side).

Further research is necessary to precisely determine the conditions that make this abstraction choice accurate. Since our modelling language supports alternative modelling choices, the reader can infer that whenever the conditions mentioned above are not satisfied, the abstraction/refinement relation on top of Figure 14 does not hold and the distribution aspect of an operation invocation has to be explicitly modelled by two separate interactions.

7 Conclusions and discussion

The approach presented in the paper results from collaboration between two Dutch research projects: Amidst [1] and Friends [4]. A common goal of these projects is to develop a middleware platform supporting (on-line) creation of services by composing (assembling) software components. The Friends project builds on an existing deployment and usage platform, which is based on the TINA architecture and on the DSC component model [2], [9]. The Amidst project develops component middleware solutions that should support flexible QoS provisioning and are largely based on OMG standards. The model-based approach presented in this paper is general, in the sense that it is independent of the middleware platform and component model being used.

The model-based part of service creation in Friends and Amidst will be supported by Testbed Studio. Current research focuses on improving and extending support for the modelling and analysis of compositions of software components. A possible extension is the conformance assessment technique presented here. This technique has been completely defined in [5], including its formal semantics. In order to demonstrate its practical use, future work will concentrate on the elaboration of case studies and the development of tools to support the technique.

References

[1] Amidst project. http://amidst.ctit.utwente.nl/.

[2] H.J. Batteram, J.-L. Bakker, J.P.C. Verhoosel, and N.K. Diakov. Design and implementation of the MESH service platform. In: *Proceedings of TINA'99 Telecommunications Information Networking Architecture Conference*, Oahu, Hawaii, USA, 12-15 april 1999.

[3] H. Eertink, W.P.M. Janssen, P.H.W.M. Oude Luttighuis, W.B. Teeuw, and C.A. Vissers, A Business Process Design Language, In: *Proceedings World Congress on Formal Methods (FM'99)*, 1999.

[4] Friends project. http://www2.telin.nl/projects/friends/home.htm.

[5] D.A.C. Quartel. *Action relations. Basic design concepts for behaviour modelling and refinement.* CTIT Ph.D-thesis series, no. 98-18. University of Twente, Enschede, The Netherlands, 1998.

[6] D.A.C. Quartel, L. Ferreira Pires, M.J. van Sinderen, H.M. Franken, and C.A. Vissers. On the role of basic design concepts in behaviour structuring. *Computer Networks and ISDN Systems 29* (1997) 413-436.

[7] D.A.C. Quartel, L. Ferreira Pires, H.M. Franken, and C.A. Vissers. An engineering approach towards action refinement. In: *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*. IEEE Computer Society Press, 1995.

[8] C. Szyperski. *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, ACM Press, New York, 1998.

[9] J.P.C. Verhoosel, M. Wibbels, H.J. Batteram, and J.L-Bakker. Rapid service development on a TINA-based service deployment platform. In: *Proceedings of TINA'99 Telecommunications Information Networking Architecture Conference*, Oahu, Hawaii, USA, 12-15 april 1999.